Tutorial Week #10 - Using Database

Topic: Integrating Hibernate with Spring Web Application (&ThymeLeaf)

Objective:

Integrate Hibernate into an existing Spring web application to enable seamless interaction with a MySQL database.

Understanding Hibernate in Spring

What is Hibernate?

Hibernate is a powerful and widely used **Object-Relational Mapping (ORM)** framework for Java. It simplifies interactions with a relational database by allowing developers to work with Java objects (entities) rather than writing SQL queries directly. Hibernate abstracts the underlying database operations, enabling easier and faster development of database-centric applications.

Key Features of Hibernate:

1. **ORM (Object-Relational Mapping):**
    - Hibernate maps Java classes to database tables and Java fields to table columns using annotations or XML configurations.
    - It allows you to persist Java objects in a database without manually writing SQL statements.
2. **Hibernate Query Language (HQL):**
    - Hibernate provides its own object-oriented query language called **HQL**, which is similar to SQL but operates on entity objects rather than database tables.
3. **Automatic Schema Generation:**
    - Hibernate can automatically create, update, or validate database schemas based on entity definitions (`@Entity` annotations).
4. **Caching:**
    - Hibernate supports first-level and second-level caching, which can significantly improve the performance of applications by reducing the number of database queries.
5. **Transaction Management:**
    - Hibernate integrates with Spring to handle transactions declaratively using annotations like `@Transactional`.
6. **Database Independence:**
    - By changing the database dialect, Hibernate can switch databases without requiring code changes.
7. **Lazy and Eager Loading:**

- Hibernate provides mechanisms to fetch data from the database either lazily (on demand) or eagerly (at once).

How Hibernate Fits into a Spring MVC Application

Spring MVC is a framework for building web applications, while Hibernate is a framework for managing database interactions. Integrating Hibernate into a Spring MVC application ensures seamless communication between the web layer and the database layer.

How They Work Together:

1. **Spring Handles the Web Layer:**
   - Spring MVC manages user requests, routes them to controllers, and renders views using Thymeleaf or JSP.
2. **Hibernate Manages the Persistence Layer:**
   - Hibernate is responsible for database-related operations such as saving, updating, deleting, and querying data.
3. **Spring ORM (Integration):**
   - Spring provides the `spring-orm` module, which integrates Hibernate with Spring's transaction management and dependency injection features.
4. **SessionFactory in Spring:**
   - Hibernate's `SessionFactory` is used for creating `Session` objects to interact with the database. Spring manages the lifecycle of the `SessionFactory` bean.
5. **Declarative Transaction Management:**
   - With Spring's `@Transactional` annotation, you can manage transactions declaratively without writing boilerplate code.

Advantages of Using Hibernate with Spring MVC

1. **Reduced Boilerplate Code:**
   - Hibernate eliminates the need to write SQL queries for CRUD operations.
   - Spring's dependency injection and transaction management reduce manual configuration.
2. **Scalability:**
   - Hibernate's caching and efficient session management allow applications to scale with ease.
3. **Portability:**
   - The database-agnostic nature of Hibernate makes it easier to switch databases by simply changing the dialect.
4. **Improved Productivity:**
   - By focusing on business logic rather than database operations, developers can build applications faster.
5. **Clear Separation of Concerns:**
   - Spring MVC handles the web layer, while Hibernate manages the persistence layer, ensuring clean architecture.

Overview of the Integration Process

1. **Add Hibernate Dependencies:**
    - Include the required Hibernate and Spring ORM dependencies in your Maven `pom.xml` file.
2. **Configure Hibernate:**
    - Set up Hibernate's `SessionFactory` and properties programmatically or using XML.
3. **Create Entity Classes:**
    - Define Java classes annotated with Hibernate's `@Entity` annotation to represent database tables.
4. **Set Up DAO Layer:**
    - Use Hibernate's `SessionFactory` to implement the data access layer for CRUD operations.
5. **Integrate with Spring Controllers:**
    - Inject DAO classes into Spring controllers to handle requests and interact with the database.
6. **Develop Views:**
    - Use Thymeleaf templates or JSP to render data fetched using Hibernate.

Why Choose Hibernate with Spring MVC?

While Spring Data JPA (a higher-level abstraction over JPA) can also be used for database operations, integrating Hibernate directly with Spring MVC is a good choice when:

- You require fine-grained control over the database interactions.
- You want to learn or leverage Hibernate-specific features such as caching, custom SQL, or native queries.

This tutorial will guide you step by step to integrate Hibernate with a Spring MVC application, enabling you to build a fully functional web application with database support.

Prerequisites before you start:

1. Basic knowledge of Java, Spring Framework, Maven, and relational databases.
2. Utilize the existing Maven project: Spring web application (includes `FrontController` and `HomeController`). You can use project from last week activity **(setting up project using thymeleaf)**
3. Ensure a MySQL database server is installed and running with the database named `springdb` containing two tables: `Customer` and `Product`.

Lessons and Steps

Lesson 1: Add Dependencies in `pom.xml`

Step 1: Add Dependencies

Open your `pom.xml` file and add the following (Hibernate and MySQL JDBC) dependencies:

Java

```
<dependencies>
    <!-- Spring Core and ORM -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.30</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.30</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>5.3.30</version>
    </dependency>

    <!-- Hibernate Core -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.15.Final</version>
    </dependency>

    <!-- MySQL Connector -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
```

```
    <!-- Thymeleaf -->
    <dependency>
        <groupId>org.thymeleaf</groupId>
        <artifactId>thymeleaf-spring5</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>
</dependencies>
```

After adding the dependencies, **right-click your project** and select **Maven > Update Project**.

Lesson 2: Hibernate Configuration

Hibernate requires configuration to connect to the database. The options are: i.declarative using xml files, ii.programmatic based using java code, and iii. using property files). In this tutorial, we will configure using programmatic approach.
Create a configuration class (HibernateConfig.java) to set up the data source, Hibernate properties, and dialect.

Step 2.a: Create Hibernate Configuration Class

Create a configuration class `HibernateConfig.java` in new folder `src/main/java/config` to set up Hibernate.

Java

```java
package config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.hibernate5.HibernateTransactionManager;
import org.springframework.orm.hibernate5.LocalSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import java.util.Properties;

@Configuration
@EnableTransactionManagement
```

```
@ComponentScan(basePackages = {"com.example.service", "com.example.entity"})
public class HibernateConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");

dataSource.setUrl("jdbc:mysql://localhost:3306/your_database_name_here"); //
Update if default port is used
        dataSource.setUsername("your_username_here");
        dataSource.setPassword("your_password_here"); // Replace with your
MySQL password
        return dataSource;
    }

    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan("com.example.entity");
        Properties hibernateProperties = new Properties();
        hibernateProperties.setProperty("hibernate.dialect",
"org.hibernate.dialect.MySQL8Dialect");
        hibernateProperties.setProperty("hibernate.show_sql", "true");
        hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "update");
        sessionFactory.setHibernateProperties(hibernateProperties);
        return sessionFactory;
    }

    @Bean
    public HibernateTransactionManager
transactionManager(LocalSessionFactoryBean sessionFactory) {
        return new HibernateTransactionManager(sessionFactory.getObject());
    }
}
```

Step 2.b: Edit your web.xml

```
Java
//no changes
```

Step 2.c: Edit your myFrontController-servlet.xml

```
Java
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- Enable annotation-driven Spring MVC -->
    <mvc:annotation-driven />

    <!-- Scan for Controllers -->
    <context:component-scan base-package="com.example.controller" />

    <!-- Scan for Configuration, Service, Repository, and Entities -->
    <context:component-scan
base-package="config,com.example.service,com.example.entity" />

    <!-- Thymeleaf View Resolver -->
    <bean id="templateResolver"

class="org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver">
        <property name="prefix" value="WEB-INF/templates/" />
        <property name="suffix" value=".html" />
        <property name="templateMode" value="HTML" />
        <property name="characterEncoding" value="UTF-8" />
    </bean>

    <bean id="templateEngine"
          class="org.thymeleaf.spring5.SpringTemplateEngine">
        <property name="templateResolver" ref="templateResolver" />
```

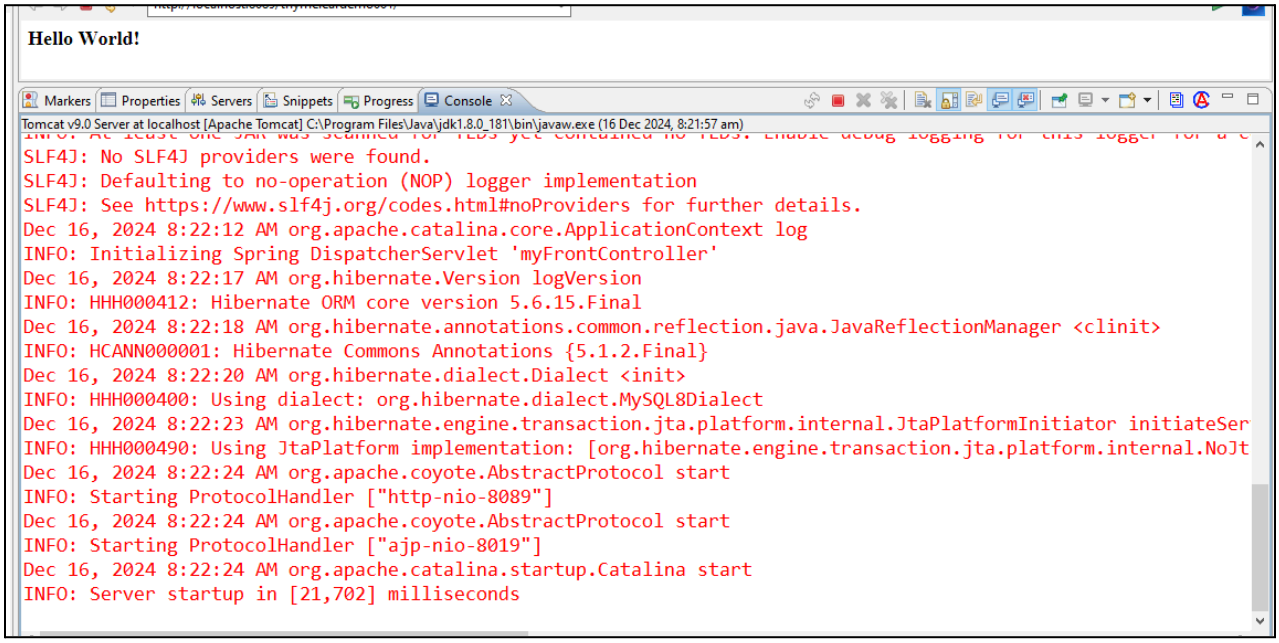
```
    </bean>

    <bean class="org.thymeleaf.spring5.view.ThymeleafViewResolver">
        <property name="templateEngine" ref="templateEngine" />
        <property name="characterEncoding" value="UTF-8" />
    </bean>

</beans>
```

Checkpoint before you proceed !!!

Run the app, and check the console log for any information/error etc …



Lesson 3: Entity Classes

Step 3: Create Entity Classes

Create entity classes for `Customer` and `Product` in the `entity` package.

Customer.java:

```java
package com.example.entity;

import javax.persistence.*;

@Entity
@Table(name = "customer")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "address")
    private String address;

    @Column(name = "contact_num")
    private String contactNum;

    @Column(name = "email")
    private String email;

    // Getters, setters, and other methods ie toString() to be added here....
}
```

Product.java:

```java
package com.example.entity;

import javax.persistence.*;
```

```
@Entity
@Table(name = "product")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "name")
    private String name;

    @Column(name = "price")
    private double price;

    // Getters, setters, toString() methods etc goes here...
}
```

Lesson 4: DAO Layer

Step 4: Create DAO Class

Create `CustomerDao.java` to handle database operations for the `Customer` entity.

Java

```
package com.example.service;

import java.util.List;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import com.example.entity.*;
@Repository
public class CustomerDao {

    private final SessionFactory sessionFactory;

    @Autowired
```

```
    public CustomerDao(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    //Complete the full 5 CRUD operations here
    public List<Customer> findAll() { // 1 - get all
        try (Session session = sessionFactory.openSession()) {
            return session.createQuery("from Customer", Customer.class).list();
        }
    }

    //2 - get by id

    //3 - create
       public void save(Customer customer) {
              // TODO Auto-generated method stub

       }

       //4 - update

       //5 - delete
}
```

Lesson 5: Integrate with Controller

Step 5: Update Controller

Modify the `HomeController.java` to use the DAO layer.

Java

```
package com.example.controller;

import com.example.entity.Customer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

import com.example.service.CustomerDao;
```

```
import java.util.List;

@Controller
@RequestMapping("/customers")
public class HomeController {

    @Autowired
    private CustomerDao customerDao;    //Dependency Injection

    @GetMapping("/list")
    @ResponseBody()
    public String listCustomers(Model model) {
        List<Customer> customers = customerDao.findAll();
        model.addAttribute("customers", customers);
         return "customer-list";
    }

    @GetMapping("/add")
    public String showAddForm(Model model) {
        model.addAttribute("customer", new Customer());
        return "customer-form";
    }

    @PostMapping("/add")
    public String addCustomer(@ModelAttribute Customer customer) {
        customerDao.save(customer);
        return "redirect:/customers/list";
    }

    @PostMapping("/delete/{id}")
    public String deleteCustomer(@PathVariable int id) {
//        customerDao.delete(id);
        return "redirect:/customers/list";
    }

  // complete the rest of the implementations here
}
```

Lesson 6: Thymeleaf Views

Step 6: Create Thymeleaf Templates

Place the following Thymeleaf files in the `src/main/resources/templates/` directory.

`customer-list.html`:

```
Unset
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Customer List</title>
</head>
<body>
<h1>Customers</h1>
<table>
    <thead>
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Address</th>
            <th>Contact</th>
            <th>Email</th>
        </tr>
    </thead>
    <tbody>
        <tr th:each="customer : ${customers}">
            <td th:text="${customer.id}"></td>
            <td th:text="${customer.name}"></td>
            <td th:text="${customer.address}"></td>
            <td th:text="${customer.contactNum}"></td>
            <td th:text="${customer.email}"></td>
        </tr>
    </tbody>
</table>
<a href="/customers/add">Add Customer</a>
</body>
</html>
```

`customer-form.html`:

```
Unset
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Add Customer</title>
```

```
</head>
<body>
<h1>Add Customer</h1>
<form th:action="@{/customers/add}" method="post" th:object="${customer}">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" th:field="*{name}" required />
    <label for="address">Address:</label>
    <input type="text" id="address" name="address" th:field="*{address}" />
    <label for="contactNum">Contact:</label>
                <input    type="text"    id="contactNum"    name="contactNum"
th:field="*{contactNum}" />
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" th:field="*{email}" />
    <button type="submit">Save</button>
</form>
</body>
</html>
```

Lesson 7: Final Steps

1. Run the application on STS/Tomcat.
2. Navigate to `/customers/list` to see the list of customers.

Tasks to complete (Monday 16 Dec 2024) - Work in Group

1. Complete all the 5 crud operations in (CustomerDao.java)
2. Complete all the implementation in CustomerController.java
3. Complete all the view pages (Thymeleaf or jsp implementation)
4. Zip and Submit the complete project source code to elearning (1 submission per group. Don't forget to write group member's name in separate text file)